Lazy pattern matching in the ML language

A. Laville

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 664

LAZY PATTERN MATCHING IN THE ML LANGUAGE

Alain LAVILLE

Mai 1987

Lazy pattern matching in the ML language

Filtrage paresseux dans le langage ML

Alain Laville

I.N.R.I.A. (Projet FORMEL) [1]
and Université de Reims[2]

[1]B.P. 105 78150 Le Chesnay CEDEX France
[2]B.P. 347 51062 Reims CEDEX France

Abstract :

This report deals with functions defined by patterns in a lazy ML system. Such a definition leads to the question of finding a lazy pattern matching algorithm. We introduce the new notion of *minimally extended pattern*, from which we derive a procedure to decide whether such an algorithm may be associated with an ordered list of patterns. Moreover this procedure gives us the means of effectively building the lazy pattern matching algorithm.

Résumé :

Ce rapport concerne la définition par filtrage des fonctions dans un système ML paresseux. Cela conduit à se poser la question de trouver un algorithme de filtrage paresseux. On introduit pour cela une nouvelle notion : celle de *motif étendu de façon minimale*, à partir de laquelle on déduit une procédure qui détermine si un tel algorithme peut être associé à une liste ordonnée de motifs. De plus cette procédure donne une construction effective de l'algorithme de filtrage paresseux.

Lazy pattern matching in the ML language

Alain Laville
I.N.R.I.A. (Projet FORMEL) *
and Université de Reims†

1 Introduction

Several of the recently developped programming languages include features that need some *pattern matching* mechanism. An important case of this use is the following : The language handles structured values and some function calls result in evaluating an expression which is determined by the way the argument (a structured value) has been built. The definition of such a function consists of couples whose first element is a structured value which may contain variables and the second one is an expression which may use the variables appearing in the first one. The meaning of such a definition is that, if the argument may be obtained by replacing the variables in the first element of a couple by suited values, then one has to evaluate the corresponding expression after doing the same replacement on the variables it contains. The first part of the couples are often called patterns and the pattern matching process determines which expression has to be evaluated. Such a mechanism may be found in languages which implement Term Rewriting Systems (such as HOPE, see [3]) but also in languages implementing Lambda Calculus (as is in particular the case of ML, see [10], or MIRANDA, see [13]).

In order to get deterministic calculations one may demand that every value can match at most one pattern in each function definition. This leads to tedious work to write such definitions and consequently this constraint is often relaxed, and, in order to maintain determinism, some rules are added which decide what expression has to be chosen when the argument matches more than one pattern. The rule used in ML is the following : the pairs (pattern, expression) are ordered and if the argument matches several patterns one uses the first of them for this ordering. Some other rules are used in different languages, for example choosing the "most defined" of the patterns that the argument matches (this implies some constraints on the set of patterns). We shall only deal in this paper with the case of the ML language.

Another important feature which is more and more implemented in programming languages is *lazyness* (also called call by need or normal order evaluation, see [11]). This means that a value is effectively computed only when it is needed to produce the result,

*B.P. 105 78150 Le Chesnay CEDEX France
†B.P. 347 51062 Reims CEDEX France

and, for a structured value, that only the needed parts are evaluated. This ensures that the language is *safe*, i.e. if computation fails there was no way to avoid this failure. This moreover gives to the language the ability of handling infinite data structures as long as only finite parts of them are used in calculations. MIRANDA and at least two implementations of ML include this feature : Lazy CAML at INRIA (see [8] or [9]) and LML the implementation of Göteborg (see [1]).

However there are problems when using pattern matching in a lazy language. One wants that the process of finding which pattern the argument matches does not force the evaluation of parts of the argument that are not needed. As far as we know this has not yet been done. For example in his paper "A Compiler for Lazy ML" [1] L. Augustsson writes :

> This rather explicit top-down, left-right ordering is perhaps unfortunate, but some ordering must be imposed to avoid the necessity of parallel evaluation of the subparts of the expression that is to be matched.

It is well known that, even in very simple cases, it is not always possible to find a lazy pattern matching process. Classical examples of this fact are the "PARALLEL OR" or Berry's example (see section 3.2.2 below). This is the problem we address in this paper and we give a (effectively computable) characterisation of the sets of patterns for which such a lazy pattern matching is possible. Moreover this characterisation yields an effective algorithm to realize the pattern matching. This solution was known in the cases where the patterns are not ambiguous (i.e. no value may match two of them), we extend it to the general case of ML patterns (ambiguous ordered list of patterns with the priority rule induced by this ordering). Moreover, the method we give here could be modified to handle other priority rules.

In order to achieve this goal, we introduce the new notion of *minimally extended pattern* (see definition 11). It will give us the key tool to check the lazyness of the pattern matching process (see section 5). It allows us to replace the set of patterns with priority by a new set in which the priority rule is captured by the syntax of the patterns. This makes the priority rule much easier to handle mechanically.

Remark 1 Along this paper, when reference to an implementation of ML is done, it is to CAML the version of ML currently implemented at INRIA (see [9] or [12]).

2 Definitions and Notations

2.1 Patterns

Definition 1 A *pattern* is a term built from the pairing operator, some constructors of (already defined) concrete data types, variables and the special symbol "_". The meaning of "_" is that one doesn't care about what may appear at the place where it is used. CAML (as other ML implementations up to date) restricts patterns to be *linear* : no variable is allowed to appear twice in the same pattern.

Definition 2 A value of CAML is said to be an *instance* of a pattern if it can be obtained from the pattern by replacing all the variables and "_" by any values (compatible with the type discipline of ML but this will always be ensured by the type-checker, so that we don't care about typing). As usual, if σ is a function which maps variables into terms, we call *substitution* the extension of σ to terms. Thus, if a term is an instance of a pattern, and if one replaces all "_" by new distinct variables, then there exists a substitution σ which yields the term as image of the pattern (that is here the classical definition of instanciation). With a list of patterns $[\ p_1, \ldots, p_n\ ]$, we shall say that a value v of CAML *matches* the pattern p_i if p_i is the *first* pattern in the list, of which v is an instance.

Definition 3 We shall say that a function is *defined by pattern* if

1. Its definition consists of an ordered list of pairs (pattern, expression)
2. Its value when applied to an argument v is obtained in the following way : first find the first pattern, say p, in the list such that v is an instance of p by a substitution σ and then evaluate the result of applying σ to the corresponding expression.

2.2 Partial Terms and Occurrences

In this paper we shall only deal with the process of finding which pattern is matched by a given value (and not of the evaluation of the corresponding expression). As patterns are linear, this implies that we don't care about the subterms corresponding to variables. Moreover, in a lazy version of CAML, at a given time a term is only partially evaluated. It will thus be convenient to define a notion corresponding to terms of which only a part is known.

We assume given the definition by pattern of a function :

$$\Pi = \left\{ \begin{array}{llll} fun & p_1 & \rightarrow & exp_1 \\ \mid & p_2 & \rightarrow & exp_2 \\ \ldots & \ldots & \ldots & \ldots \\ \mid & p_n & \rightarrow & exp_n \end{array} \right.$$

where the p_i's are the patterns to be matched against the value to which one applies the function.

One defines a signature Σ containing all the constructors that appear in the patterns $p_1, \ldots, p_n$ and another symbol Ω which will denote the "unknown" or "undefined". As we want to extend methods of Terms Rewriting Systems theory which consider patterns as terms represented by trees, we shall sometimes have to use this representation. In such cases we need a symbol to be placed at the root of the tree (it will denote the ML function which is to apply). So we add to Σ a new symbol $\mathcal{F}$ which when necessary denotes this function. We assume given a set of variables $\mathcal{V}$ containing all the variables that appear in the patterns $p_1, \ldots, p_n$. If one replaces the "_" of the p_i's by fresh distinct variables taken from $\mathcal{V}$, then all the $\mathcal{F}(p_i)$ are terms of the algebra built over $\Sigma \cup \mathcal{V}$.

Definition 4 We shall call *partial term* every term built over Σ. We shall only use *term* to denote a partial term in which there is no symbol Ω (but we do not forbid to use "partial term" even in this case).

Partial terms provides us with a formalism suited as well for patterns (which are partially undefined terms) and for lazy values (which may be thought of as partially unknown since they are not completely evaluated). To avoid introducing new notation, we shall denote from now on by p_i the partial term obtained from $\mathcal{F}(p_i)$ by replacing all variables and "_" by Ω's. We shall denote by Π in the following the ordered sequence : $[\, p_1, \ldots, p_n \,]$ (they are the p_i's we just defined).

We define a partial ordering (denoted by $\leq$) over the set of all partial terms as follows :

- For each partial term M : $\Omega \leq M$
- $F(M_1, \ldots, M_n) \leq F(N_1, \ldots, N_n)$ if and only if $M_i \leq N_i$ $(1 \leq i \leq n)$

We shall use the notation $M \uparrow N$ when M and N have a common upper bound (and we shall say that M and N are compatible), and the notation $M \sharp N$ if they don't have one (and we shall say that M and N are incompatible).

The ordering $\leq$ is a kind of prefix ordering with the meaning that a partial term is less than another if it is less defined (or less known).

Remark 2 Assume that a partial term M (representing here a lazy value of CAML) given as argument to $\mathcal{F}$ is sufficiently defined to decide which of the right hand side expressions defining $\mathcal{F}$, say exp_{i_0}, has to be evaluated after instantiation to return the desired value of the application. Then it is clear that the following must hold : $p_{i_0} \leq M$. The converse is true if and only if for each pair (p_i, p_j) of patterns one has $p_i \sharp p_j$. The *if* part is obvious. For the *only if* part, assume $p_i \uparrow p_j$ with $i < j$ and let M be a common upper bound of p_i and p_j then, although $p_j \leq M$, exp_j is not the expression to evaluate.

Definition 5 When seeing partial terms as trees, we shall say that M_i is the i^{th} *son* of the partial term $F(M_1, \ldots, M_n)$. We call *occurrence* an integer list which designates a subterm of a given partial term. For example, the occurrence [2; 3] points to the third son of the second son of the full partial term. The prefix ordering of occurrences will be denoted by $\leq$. For a given partial term M, we shall denote $\mathcal{O}(M)$ the set of all occurrences in M, $\overline{\mathcal{O}}(M)$ the set of occurrences in M where the symbol is not Ω and $\mathcal{O}_\Omega(M)$ the set of occurrences in M where the symbol is Ω. The symbol in M at occurrence u will be denoted by $M(u)$.

2.3 Matching Predicates

We shall now define some predicates over the set of partial terms (i.e. functions with values in the set {tt, ff} of the truth values).

Definition 6 For each $i \in \{1, \ldots, n\}$, the predicate $match_i$ is defined by $match_i(M) = \mathrm{tt}$ if and only if the following two conditions hold :

1. $p_i \leq M$
2. $\forall j < i \quad p_j \sharp M$

Remark 3 We recall that the patterns are ordered, and that matching searches for the *first* suitable pattern in the list (see definition 2). The meaning of the preceding definition is that $match_i(M) = \mathrm{tt}$ iff M is sufficiently defined to decide that if M is the argument of $\mathcal{F}$ then one has to evaluate (after suitable instantiation) the expression number i.

Definition 7 We now define the predicate $match_\Pi$ by $match_\Pi(M) = \mathrm{tt}$ if and only if $match_i(M) = \mathrm{tt}$ for some $i \in \{1, \ldots, n\}$.

Remark 4 The meaning is of course that M is sufficiently defined to decide which right hand side has to be evaluated to get the value of $\mathcal{F}(M)$.

3 Properties of the Match Predicates

3.1 Monotonicity

The following lemma groups some useful characterizations and properties related with the ordering of partial terms.

Lemma 1

a) *$M \sharp N$ if and only if there exists an occurrence u in $\overline{\mathcal{O}}(M) \cap \overline{\mathcal{O}}(N)$ such that $M(u) \neq N(u)$.*

b) *$M \leq N$ if and only if for each $u \in \mathcal{O}(M)$ either $M(u) = \Omega$ or $M(u) = N(u)$.*

c) *$M \uparrow N$ if and only if the two following hold :*

- *$\forall u \in \mathcal{O}(M)$ either $M(u) = N(u)$ or $\exists v \leq u$ such that $N(v) = \Omega$*
- *$\forall u \in \mathcal{O}(N)$ either $N(u) = M(u)$ or $\exists v \leq u$ such that $M(v) = \Omega$*

d) *The ordering $\leq$ over the partial terms is well founded (i.e. there exists no infinite strictly decreasing sequence).*

e) *If $match_i(M) = \mathrm{tt}$ then for all $j \neq i$ $match_j(M) = \mathrm{ff}$.*

Proof : a), b), c) and d) are straightforward. To get e) one simply remarks that if $j < i$ then $M \sharp p_j$, and if $j > i$ then one can not have $M \sharp p_i$, in both cases $match_j(M)$ is false. ■

Definition 8 We order the set of truth values by defining $\mathrm{ff} < \mathrm{tt}$. Using the ordering on partial terms this allows to define *increasing* predicates.

Proposition 1 *The predicates $match_i$ and the predicate $match_\Pi$ are all increasing.*

Proof : Assume that $match_i(M) = \mathrm{tt}$ and let $N \geq M$. By definition of $match_i$ one has $M \geq p_i$ and hence $N \geq p_i$. From the same definition we know that $M \sharp p_j$ for each $j < i$. Thus (lemma 1 a) $\forall j < i \; \exists u_j \in \overline{\mathcal{O}}\ (M)$ such that

$$M(u_j) \neq \Omega \ , \ p_j(u_j) \neq \Omega \text{ and } p_j(u_j) \neq M(u_j)$$

Since, from part b) of lemma 1, $N(u_j) = M(u_j)$, the result follows obviously for each predicate $match_i$.

It is now straightforward, from the definition, to reduce the case of $match_\Pi$ to the preceding one. ■

3.2 Sequentiality

3.2.1 Definition

We define here a formal property of increasing predicates, called sequentiality and due to G. Kahn and G. Plotkin (see [7]), and also to G. Berry and P.L. Curien (see [5]) which is strongly connected with the problem we address as will be shown later.

Definition 9 Definition is given in three steps as follows :

1. For a given partial term M and an increasing predicate P, we shall say that an occurrence u in M is an *index* of P in M if the three following conditions hold :
   a) $M(u) = \Omega$
   b) $P(M) = \text{ff}$
   c) $\forall N \geq M$ if $P(N) = \text{tt}$ then $N(u) \neq \Omega$
2. For a given partial term M and an increasing predicate P, we shall say that P is *sequential at* M if and only if the two conditions $P(M) = \text{ff}$ and there exists $N \geq M$ such that $P(N) = \text{tt}$ imply together that P has an index in M.
3. Finally we shall say that an increasing predicate P is *sequential* if it is sequential at every partial term.

3.2.2 Examples

We shall give examples, with the predicates $match_\Pi$ associated with various lists of patterns (the algebra Σ is built each time according to section 2.2).

1. With $\Pi = [\mathcal{F}(\text{a,a}); \mathcal{F}(\text{a,b}); \mathcal{F}(\text{b,a}); \mathcal{F}(\text{b,b})]$, if $match_\Pi(M) = \text{ff}$ and there exists $N \geq M$ such that $match_\Pi(N) = \text{tt}$, then M must belong to the set $\{\Omega, \mathcal{F}(\Omega,\Omega), \mathcal{F}(\Omega,\text{a}), \mathcal{F}(\Omega,\text{b}), \mathcal{F}(\text{a},\Omega), \mathcal{F}(\text{b},\Omega)\}$. It is easy to check that all the occurrences where Ω appears are indexes for $match_\Pi$, and thus that this predicate is sequential.

2. With $\Pi = [\mathcal{F}(\text{a},\Omega); \mathcal{F}(\Omega,\text{b})]$, if $match_\Pi(M) = \text{ff}$ and there exists $N \geq M$ such that $match_\Pi(N) = \text{tt}$, then M must belong to the set $\{\Omega, \mathcal{F}(\Omega,\Omega), \mathcal{F}(\text{b},\Omega), \mathcal{F}(\Omega,\text{b})\}$. The fourth element of this set may look rather surprising since it is one of the patterns but it is an example of the fact that $match_i(p_i)$ may be false.

   The predicate $match_\Pi$ has trivially an index at Ω. One easily sees that if $match_\Pi(N) = \text{tt}$ then $N = \mathcal{F}(N_1, N_2)$ with $N_1 \neq \Omega$. Hence $match_\Pi$ has an index at $\mathcal{F}(\Omega,\Omega)$ and $\mathcal{F}(\Omega,\text{b})$. If $N \geq \mathcal{F}(\text{b},\Omega)$ then $match_1(N) = \text{ff}$ so that if $match_\Pi(N) = \text{tt}$ $match_2(N) = \text{tt}$ must hold. Hence $N([2]) = \text{b}$ and $[2]$ is an index of $match_\Pi$ at $\mathcal{F}(\text{b},\Omega)$ (see notations in definition 5) : $match_\Pi$ is sequential.

3. With $\Pi = [\mathcal{F}(\text{a,b}); \mathcal{F}(\Omega,\Omega)]$, if $match_\Pi(M) = \text{ff}$ and there exists $N \geq M$ such that $match_\Pi(N) = \text{tt}$, then M must belong to the set $\{\Omega, \mathcal{F}(\Omega,\Omega), \mathcal{F}(\text{a},\Omega), \mathcal{F}(\Omega,\text{b})\}$. One easily checks that $match_\Pi$ has an index at Ω, $\mathcal{F}(\text{a},\Omega)$ and $\mathcal{F}(\Omega,\text{b})$. It has not at $\mathcal{F}(\Omega,\Omega)$ since both $\mathcal{F}(\text{b},\Omega)$ and $\mathcal{F}(\Omega,\text{a})$ verify $match_2$ and hence verify $match_\Pi$ too.

4. We give now a version of what is known as Berry's example : $\Pi = [\mathcal{F}(\text{true,false},\Omega);$ $\mathcal{F}(\text{false},\Omega,\text{true});$ $\mathcal{F}(\Omega,\text{true,false})]$. One sees that there is no index for $match_\Pi$ at $\mathcal{F}(\Omega,\Omega,\Omega)$: since the patterns are incompatible, they all verify $match_\Pi$ and one can find an Ω at every occurrence.

3.2.3 Sequentiality and Lazyness

We address now the question of choosing the right hand side when one needs to evaluate $\mathcal{F}(v)$ in a lazy version of ML, where v is any value (and hence may be only partially evaluated). We want that this process does not force the evaluation of any part of v which is not necessary to make the choice.

Definition 10 We call *pattern matching algorithm* any deterministic algorithm which matches any partial term against Π (see definition 2). As partial terms are trees and a pattern is a prefix of any partial term that matches it, this process has to work in a top-down way.

We say that a pattern matching algorithm is *lazy* if it satisfies the preceding condition. We may express this constraint in the following way : Let U be the set of all occurrences in v where the symbol was evaluated during the pattern matching process. Denote v_Ω the partial term which coincides with v along U and is completed with Ω's according to the arities of the symbols used. Then we ask v_Ω to be less than or equal to (for the ordering of partial terms) every prefix of the full value v which is sufficient to choose the right hand side.

This property of lazy pattern matching is connected with sequentiality, as shows the following theorem.

Theorem 1 *Given a function defined by pattern, there exists an associated lazy pattern matching algorithm if and only if the predicate $match_\Pi$ is sequential.*

Proof : We shall only give a sketch of the proof.

Assume that the predicate $match_\Pi$ is sequential. Then it suffices at each step of the pattern matching process, to look in v at an occurrence that is an index for $match_\Pi$ in the prefix of v that was already explored.

Conversely, assume $match_\Pi$ not to be sequential and let M be a partial term with no index. Let run the pattern matching algorithm from Ω, getting the symbol at u in M as long as it looks at an occurrence $u \in \overline{\mathcal{O}}(M)$. Let u_0 the first occurrence in $\mathcal{O}_\Omega(M)$ which the algorithm will look at. Since there is no index in M one can find a partial term N such that $\mathrm{N}(u_0) = \Omega$ and $match_\Pi(N) = \mathrm{tt}$. For this N the algorithm makes useless work, or in other words it may fail to recognize a matching by failing during the evaluation of a not needed part of the argument. ■

4 Equivalent Matching Predicate

4.1 Introduction

The sequentiality of the predicate $match_\Pi$ is not so easy to test. So we shall give an equivalent definition of this predicate the sequentiality of which can be studied in a more tractable way.

The idea is as follows : If $match_i(M) = \mathrm{tt}$, then M has to be incompatible with all the patterns p_j such that $j < i$. Thus we shall try to replace the single pattern p_i by a *set* of patterns, each of them greater than p_i and incompatible with all the preceding patterns. Then we shall replace the initial list of patterns by the longer one that we get from the replacement of each pattern by the set above. This new list may be constituted of pairwise incompatible patterns, and the matching against it may be studied by already known methods. The list may also contain two (or more) compatible patterns, and there is no sequential pattern matching algorithm for the initial definition (i.e. $match_\Pi$ is not sequential) as will be proved later.

Of course we have to show that we did not change the function call when modifying the set of patterns. Moreover, we have to take care that the *new* match predicates have the same sequentiality properties as the old ones. The trouble is the following : Assume we deal with the function AND over the booleans which may be defined by pattern matching with two cases, namely

$$\begin{array}{lll} fun\ (true, true) & \rightarrow & true \\ \mid\ \ (x, y) & \rightarrow & false \end{array}$$

One can see that there is no sequential pattern matching in this case using the same argument as in the third example of 3.2.2. We could define the same function over the set of boolean values with incompatible patterns, for example by

$$\begin{array}{lll} fun\ (true, true) & \rightarrow & true \\ \mid\ \ (false, x) & \rightarrow & false \\ \mid\ \ (true, false) & \rightarrow & false \end{array}$$

But although this is the same function over the booleans, we can easily see that it is not equivalent to the first one in a lazy system. Assume we give it as argument a pair the first part of which fails to evaluate and the second evaluates to false, then the first function returns false and the last one fails. Expanding the pattern (x, y) to the set $\{(false, x); (true, false)\}$ introduced a precedence of the first part of the couple which was not in the first definition (one may check that there exists in this case a lazy pattern matching algorithm : look first at first occurrence).

4.2 Extended Patterns

In the following lemma,we use the notation OCC_Π to denote the set of occurrences which may be useful to choose the pattern at least in one case, i.e.

$$OCC_\Pi = \bigcup_{i=1}^{n} \overline{\mathcal{O}}(p_i)$$

Lemma 2 *If $match_\Pi(M) = \mathrm{tt}$, there exists a prefix M' of M such that :*

1. $match_\Pi(M') = tt$
2. $\overline{\mathcal{O}}(M') \subset OCC_\Pi$

Proof : If u is an occurrence in M not belonging to OCC_Π, there exists a maximal prefix of u, say v, in OCC_Π. In order to get M' we cut M at every such occurrence v and complete with Ω's according to the arities of remaining symbols. We easily get the property $match_\Pi(M') = \mathrm{tt}$, since, by lemma 1, we express it using only symbols at occurrences member of OCC_Π and these symbols are the same in M and M'. ■

Definition 11 We shall call *minimally extended pattern* (associated with Π) any partial term t verifying the following two properties :

1. $match_\Pi(t) = \mathrm{tt}$
2. $\forall t' < t, match_\Pi(t') = \mathrm{ff}$

We shall denote the set of all minimal extended patterns by MEP_Π.

Proposition 2 *The set MEP_Π is a finite set. Furthermore it is possible to effectively build this set from the initial list of patterns.*

Proof : From the second part of the definition, for each minimally extended pattern t, one has $\overline{\mathcal{O}}(t) \subset OCC_\Pi$. Since we deal with (partial) terms over a finite signature, the results follow (all the calculations involved are finite). ■

4.3 Examples :

1. In the case of the AND function used above (see section 4.1), the set MEP_Π contains the couple (true,true) coming from the first pattern, and the two couples (false,Ω) and (Ω,false) coming from the second pattern. One sees here a case where two elements of MEP_Π are compatible. As we said above, this is a case where no sequential pattern matching is possible. When we used this function (in the introduction above) the trouble with the sequentiality came from the following : we replaced then the second pattern by the set {(false,Ω); (true,false)}. In this set the second element is not a *minimally* extended pattern : it asks for too much information about the value.

2. If one wants to use the classical PARALLEL OR, and tries to define it as :

$$\begin{array}{lrcl} fun & (true, x) & \rightarrow & true \\ | & (x, true) & \rightarrow & true \\ | & (x, y) & \rightarrow & false \end{array}$$

   the set MEP_Π will be {(true,Ω);(false,true);(false,false)}. This set contains only pairwise incompatible patterns, which correspond one to one to the initials ones. We shall discuss the meaning of that fact after giving the procedure to study the existence of a lazy pattern matching algorithm.

3. In the case of Berry's example (see 3.2.2), since the three patterns are pairwise incompatible, the set MEP_Π simply contains them.

4. We give now some abstract examples, using a signature Σ consisting of $\mathcal{F}$, Ω and three symbols (denoted a, b and c) each of arity zero. We shall not discuss here the question of sequentiality. We shall do after giving the procedure to decide it.

   - With initial patterns $p_1 = \mathcal{F}(a,\Omega,\Omega)$, $p_2 = \mathcal{F}(\Omega,b,\Omega)$ and $p_3 = \mathcal{F}(\Omega,\Omega,c)$, p_1 remains unmodified ; p_2 has to be replaced by $\{\mathcal{F}(b,b,\Omega), \mathcal{F}(c,b,\Omega)\}$ meaning that it can be recognized only after excluding a symbol "a" in first place ; similarly p_3 is replaced by the set $\{\mathcal{F}(b,c,c), \mathcal{F}(c,c,c), \mathcal{F}(b,a,c), \mathcal{F}(c,a,c)\}$ since one has to exclude at the same time "a" in first place and "b" in second place.
   - With the list of patterns $[\mathcal{F}(a,\Omega,\Omega); \mathcal{F}(b,\Omega,\Omega); \mathcal{F}(\Omega,\Omega,c)]$, the first two are not modified, and the third is replaced by $\{\mathcal{F}(c,\Omega,c)\}$.
   - With the list of patterns $[\mathcal{F}(a,\Omega,\Omega); \mathcal{F}(a,b,\Omega); \mathcal{F}(\Omega,\Omega,c)]$, the first one remains, the second is discarded (replaced by the empty set) and the third is replaced by $\{\mathcal{F}(c,\Omega,c), \mathcal{F}(b,\Omega,c)\}$.
   - With the list of patterns $[\mathcal{F}(a,b,\Omega); \mathcal{F}(\Omega,a,\Omega); \mathcal{F}(\Omega,\Omega,c); \mathcal{F}(\Omega,\Omega,\Omega)]$ the first two remain, the third is replaced by the set $\{\mathcal{F}(\Omega,c,c), \mathcal{F}(b,b,c), \mathcal{F}(c,b,c)\}$, and the last one by the set $\{\mathcal{F}(\Omega,c,a), \mathcal{F}(\Omega,c,b), \mathcal{F}(b,b,a), \mathcal{F}(b,b,b), \mathcal{F}(c,b,a), \mathcal{F}(c,b,b)\}$.
   - We slightly change the list of patterns (only modifying the first symbol in the second pattern) getting the list $[\mathcal{F}(a,b,\Omega); \mathcal{F}(a,a,\Omega); \mathcal{F}(\Omega,\Omega,c); \mathcal{F}(\Omega,\Omega,\Omega)]$. Now the first two remain, the third is replaced by $\{\mathcal{F}(\Omega,c,c), \mathcal{F}(b,\Omega,c), \mathcal{F}(c,\Omega,c)\}$ and the last one by the set $\{\mathcal{F}(\Omega,c,a), \mathcal{F}(\Omega,c,b), \mathcal{F}(b,\Omega,a), \mathcal{F}(b,\Omega,b), \mathcal{F}(c,\Omega,a), \mathcal{F}(c,\Omega,b)\}$.

4.4 New Definition of Matching Predicate

Definition 12 We call $match'_\Pi$ the predicate defined over the set of partial terms by $match'_\Pi(M) = \mathrm{tt}$ if and only if there exists an element t of MEP_Π such that $t \leq M$.

Proposition 3 *The two predicates $match'_\Pi$ and $match_\Pi$ are the same predicate over the set of all partial terms.*

Proof : Assume $match'_\Pi(M) = \mathrm{tt}$, and let $t \in MEP_\Pi$ be such that $t \leq M$. From the definition of MEP_Π, one has $match_\Pi(t) = \mathrm{tt}$. Since this predicate is increasing (proposition 1) we get $match_\Pi(M) = \mathrm{tt}$.

Conversely, the result is straightforward since the ordering on partial terms is well founded (lemma 1 d). We may remark that nothing here ensures that for each partial term M verifying $match_\Pi(M) = \mathrm{tt}$, there exists a unique $t \in MEP_\Pi$ such that $t \leq M$. In many cases this result will not hold. ■

Corollary 1 *The predicate $match_\Pi$ is sequential if and only if $match'_\Pi$ is.*

5 Deciding Sequentiality

5.1 Incompatible Minimally Extended Patterns

When all the minimally extended patterns are pairwise incompatible, the sequentiality of $match'_\Pi$ is easily decided using already known methods. One only has to check if this predicate is sequential at every partial term which is a prefix of an element of MEP_Π. Moreover, one can exhibit a lazy pattern matching algorithm when the checking succeeds. These two goals are achieved by trying to build a "matching tree" (see for details Huet and Lévy [6] where this method is introduced). A matching tree is a tree of which each node contains a partial term M with an index u of $match'_\Pi$ in M, and the branches issued from a node are labelled with the symbols that may be placed at u in M in order to get a match. Leaves correspond to elements of MEP_Π which mean a success in the matching process. The root of the tree contains the partial term Ω with the trivial index of $match'_\Pi$ at this partial term.

Given the matching tree, the lazy pattern matching algorithm is as follows : start at the root of the tree and when reaching a node look in the value at the occurrence contained in the node ; follow the branch of which the label is the symbol found in the value if there is one (if there is no such branch the matching process fails) ; when reaching a leave one ensures that the value is greater than one of the elements of MEP_Π, say p, the one corresponding to the leave. Hence the value matches the unique initial pattern p_i such that $match_i(p) = \mathrm{tt}$.

5.2 Compatible Minimally Extended Patterns

We deal now with the case where there exists two (or more) compatible extended minimal patterns.

Lemma 3 *Let s and t be two compatible minimally extended patterns. By definition both verify the predicate $match_\Pi$. Hence there exists i and j such that $match_i(s) = \mathrm{tt}$ and $match_j(t) = \mathrm{tt}$. Then $i = j$.*

Proof : Let M be a partial term such that $s \leq M$ and $t \leq M$ (M exists since s and t are compatible). As all the predicates $match_k$ are increasing (proposition 1) one has $match_i(M) = \mathrm{tt}$ and $match_j(M) = \mathrm{tt}$. Since, for a given partial term, only one of the $match_k$ may be true (lemma 1 e) we get $i = j$. ■

Examples :

It is easy to see from the examples above (section 4.3), that one effectively can get compatible patterns and that the preceding proposition holds in these cases.

Proposition 4 *If MEP_Π contains two (or more) compatible patterns then the predicate $match_\Pi$ is not sequential. Hence there is no lazy pattern matching algorithm for the initial list of patterns.*

Proof : Assume that there exists two compatible minimally extended patterns s and t. As they are both minimal they are not comparable for the ordering over partial terms. Hence their greatest lower bound M (which always exists) is strictly less than both s and t. Hence one must have $match_\Pi(M) = \mathrm{ff}$ (from the definition of minimally extended pattern).

We shall prove that there exists no index for $match_\Pi$ in M. Let u be an occurrence of Ω in M and look at $s(u)$ and $t(u)$. There are two cases to consider :

- If s and t have the same symbol at occurrence u, this symbol must be Ω. Otherwise replacing the Ω at u in M by the same symbol as in s and t (extended with Ω's according with its arity) would give us a partial term less than s and t and strictly greater than M. This would contradict the assumption that M is the greatest lower bound of s and t.

- If s and t have distinct symbols at occurrence u, one of them has to be Ω. This is a consequence of the caracterisation of compatible partial terms given in lemma 1.

In all cases we get a partial term greater than M, for which the predicate $match_\Pi$ returns tt, and having a symbol Ω at occurrence u. Hence this occurrence is not an index of $match_\Pi$ in M. As this is true for every occurrence of Ω in M $match_\Pi$ is not sequential at M. ■

Theorem 2 *The existence of a lazy pattern matching algorithm for a given list of patterns is decidable. Moreover we are able to effectively build such an algorithm if there exists one.*

Proof : Using the preceding results, the procedure is as follows : Build MEP_Π and check for compatible patterns in this set. If one can find such patterns, there is no lazy pattern matching algorithm. If such patterns do not exist one only has to try to build a matching tree. If this building succeeds it gives a lazy pattern matching algorithm, if it fails the predicate $match_\Pi$ is not sequential (the only failure may come from the lack of an index in one of the partial terms that are placed in the tree) and hence there exists no lazy pattern matching algorithm. ■

5.3 Examples

We shall look now at the examples of section 4.3 from the point of view of lazy pattern matching.

- The set MEP_Π associated with the AND function contains two compatible patterns (both coming from the second rule, see section 4.3), hence there is no lazy pattern matching algorithm. In fact this is the obviously correct result, since the meaning of the function is : If either of the two arguments is "false" then the result is false. In an abstract sense this is exactly the PARALLEL OR (exchanging false and true) for which we expect no sequentiality to hold.

- We tried to define a PARALLEL OR by

$$\begin{array}{lll} fun & (true, x) & \rightarrow \ true \\ | & (x, true) & \rightarrow \ true \\ | & (x, y) & \rightarrow \ false \end{array}$$

  and got $\{\mathcal{F}$(true,Ω); $\mathcal{F}$(false,true); $\mathcal{F}$(false,false)$\}$ as MEP_Π. In order to determine if a lazy pattern matching algorithm exists one has to build a matching tree. Starting from $\mathcal{F}(\Omega, \Omega)$, one has to look at its first argument. We can find here two symbols : getting true we match the first rule, getting false we have to look to the second argument; getting here true we match the second rule, getting false we match the third one. This means that there is here a sequential pattern matching algorithm. This is because the function we defined is not a parallel OR. In fact, due to the priority rule of ML when choosing which pattern the value matches, the function is OR(x,y) = if x then true else y.

- In Berry's example, although the initial patterns were pairwise incompatible (and hence the set MEP_Π too), one cannot find a lazy pattern matching algorithm since there is no index for $match_\Pi$ in $\mathcal{F}(\Omega,\Omega,\Omega)$. In fact, whatever the first place to look at would be, one can build a term M such that M matches one of the patterns and the value in M at that place is irrelevant.

- In the first abstract example, we have Π = [$\mathcal{F$(a,Ω,Ω); $\mathcal{F$(Ω,b,Ω); $\mathcal{F$(Ω,Ω,c)] and MEP_Π = $\{\mathcal{F}$(a,Ω,Ω), $\mathcal{F}$(b,b,Ω), $\mathcal{F}$(c,b,Ω), $\mathcal{F}$(b,c,c), $\mathcal{F}$(c,c,c), $\mathcal{F}$(b,a,c), $\mathcal{F}$(c,a,c)$\}$. All the patterns are pairwise incompatible. The matching tree is easily build : look at the first occurrence, if you find "a" then rule 1, else look at occurrence 2, if you find "b" then rule 2, else look at occurrence 3, if you find "c" then rule 3 else match fails.

- With Π = [$\mathcal{F$(a,Ω,Ω); $\mathcal{F$(b,Ω,Ω); $\mathcal{F$(Ω,Ω,c)] and MEP_Π = $\{\mathcal{F}$(a,Ω,Ω), $\mathcal{F}$(b,Ω,Ω), $\mathcal{F}$(c,Ω,c)$\}$ the patterns are pairwise incompatible and the building of the matching tree is straightforward.

- With Π = [$\mathcal{F$(a,Ω,Ω); $\mathcal{F$(a,b,Ω); $\mathcal{F$(Ω,Ω,c)] and MEP_Π = $\{\mathcal{F}$(a,Ω,Ω), $\mathcal{F}$(b,Ω,c), $\mathcal{F}$(c,Ω,c)$\}$ the patterns are pairwise incompatible and the building of the matching tree is straightforward.

- With Π = [$\mathcal{F$(a,b,Ω); $\mathcal{F$(Ω,a,Ω); $\mathcal{F$(Ω,Ω,c); $\mathcal{F$(Ω,Ω,Ω)], the set MEP_Π will consist of $\{\mathcal{F}$(a,b,Ω), $\mathcal{F}$(Ω,a,Ω), $\mathcal{F}$(Ω,c,c), $\mathcal{F}$(b,b,c), $\mathcal{F}$(c,b,c), $\mathcal{F}$(Ω,c,a), $\mathcal{F}$(Ω,c,b), $\mathcal{F}$(b,b,a), $\mathcal{F}$(b,b,b), $\mathcal{F}$(c,b,a), $\mathcal{F}$(c,b,b)$\}$. One can easily check that the extended patterns are pairwise incompatible and it remains to build the matching tree. In $\mathcal{F}(\Omega,\Omega,\Omega)$ the index is the second occurrence. If we find here the symbol "a" then we have to apply the second rule. If we find a "b" we have to look at the first occurrence : if there is a "a" apply the first rule, otherwise look at the third occurrence where a "c" leads to rule three and "a" or "b" to rule four. If, when looking at the second occurrence, one finds a "c" then one has to look at the third occurrence where a "c" leads to the third

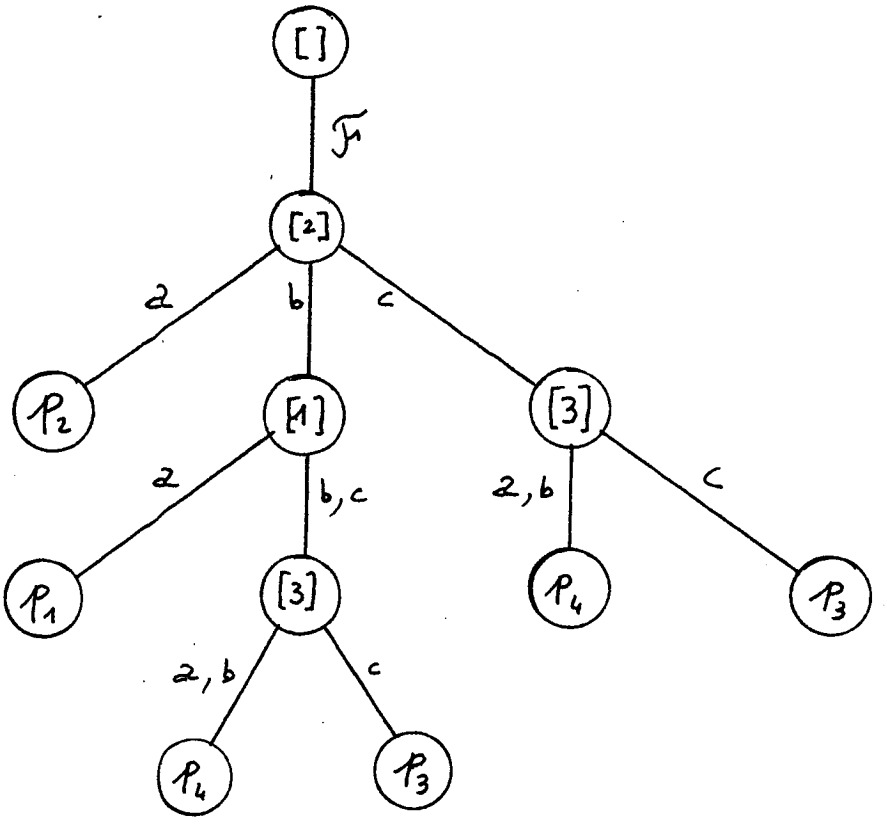

Figure 1: Matching tree

rule and "a" or "b" to the fourth. (see the matching tree, with some modifications in order to improve readability, in figure 1).

Since the building of the matching tree succeeds, we get a lazy pattern matching algorithm. However this algorithm is not straightforward and some ingenuity would be needed for a compiler to derive it directly from the initial list of patterns.

- With $\Pi = [\mathcal{F}(a,b,\Omega); \mathcal{F}(a,a,\Omega); \mathcal{F}(\Omega,\Omega,c); \mathcal{F}(\Omega,\Omega,\Omega)]$ the set MEP_Π will consist of $\{\mathcal{F}(a,b,\Omega), \mathcal{F}(a,a,\Omega), \mathcal{F}(c,\Omega,c), \mathcal{F}(b,\Omega,c), \mathcal{F}(\Omega,c,c), \mathcal{F}(b,\Omega,a), \mathcal{F}(b,\Omega,b), \mathcal{F}(c,\Omega,a), \mathcal{F}(c,\Omega,b), \mathcal{F}(\Omega,c,a), \mathcal{F}(\Omega,c,b)\}$. In this case there are compatible minimally extended patterns (for example $\mathcal{F}(c,\Omega,c)$ and $\mathcal{F}(\Omega,c,c)$ in the set replacing the third initial pattern) and we cannot find a lazy pattern matching algorithm. A trouble for example comes from the fact that in order to recognize the third rule, one has to find a "c" at the third occurrence and to discard the first two rules. But this second condition is achieved looking at the first occurrence in some terms and at the second in others and the occurrence to look at can not be choosen before looking at its symbol.

  The very important difference between the last two examples is due to a little change in the set of initial patterns. It would be rather difficult to detect such differences

without a mechanical process such as the one we give here.

- We end with a more realistic example. It is an ML function which simulates a piece of the Categorical Abstract Machine (C.A.M.).

```
let rec Exec = fun
  (pair(x,y), (car::CC), D) ->  Exec (x, CC, D)
| (pair(x,y), (cdr::CC), D) ->  Exec (y, CC, D)
| (T, (Cons::CC), (val(T')::D))
            ->  Exec (pair(T',T), CC, D)
| (pair(closure(x,y),z), (app::CC), D)
            ->  Exec (pair(y,z), x, (adr(CC)::D))
| (T, (cur(x)::CC), D)
            ->  Exec (closure(x,T), CC, D)
| (T, [], _) -> T;;
```

One can see the first element of each pattern as the register of the machine, the second one as the code (list of instructions) to be executed and the third one as a stack (implemented as a list). The above function describes what is to do when getting each instruction at first place in the code or when code is empty. Of course in a real case there would be much more instructions and hence much more cases in the function definition.

In this case, we see that all the patterns are pairwise incompatible : they are all distinct in the code argument. However, what is interesting here is that the building of the matching tree shows that one has to look first at the second argument and if it is not the empty list to the first element of the list that one finds. This is sufficient to select the only rule which may be applied (it remains to verify the other conditions to apply it). All the already implemented pattern matching compilations fail to recognize this fact and give a code that does useless work (for the algorithms used see [1], [4] and [12]).

6 Conclusion

6.1 Implementation

In order to insert our method in a compiler for an ML system, one has to solve some other problems : for example ML accepts infinite signatures to build the patterns, allowing them to contain integers or strings. This may be solved since there is only a finite number of integers or strings that effectively appear in the patterns ; one only has to group other values as a single otherwise case. The same method is used to take into account that other constructors may appear in the value to match, than those that one find in the patterns. Some trouble may come from the internal representation of ML structured values, of which some parts are discarded for the sake of efficiency (see Suarez [12]) : this only leads to

some complications in the algorithm. We also have to give a compilation of the pattern matching even in the cases where there is no lazy pattern matching algorithm : issuing a warning, we pick an occurrence that is useful at least in one case, and continue the building of a matching tree with this occurrence as if it was an index. This gives a pattern matching algorithm but of course lazyness is lost.

We are also led to address the point of efficiency in our compilation, and particularly when searching for indexes and building the set MEP_{Π}. However experimental versions of a pattern matching compiler produce CAM code from ML code using less than twice the time that the present compiler uses. On the other hand these versions use simpler internal representations of the patterns and one can expect some gain on the parsing time. Moreover the CAM code seems to be more compact and may run faster. Another possible improvement (that is still under test too) is the following : when our algorithm recognizes the expression to evaluate, the set of occurrences it looked at in the argument is known ; if we keep pointers at them the access to variables appearing in this expression may be significantly improved.

All these facts lead us to try to implement such a pattern matching compilation not only in the lazy version of the CAML system but also in the strict version. We expect some efficiency gain in the two cases (we recall here the example of the function simulating the C.A.M.). However some further experiment is needed to determine exactly what is really useful among the possible improvements we listed above.

6.2 Further work

Apart from the implementation work in progress, we want to look at some extensions of this paper.

The first one is to extend the theoretical results to other systems. This covers ambiguous pattern matching with other priority rule than ML, but also investigating the possibilities of extending them to Term Rewriting Systems with ambiguous left-hand-sides and priority meta-rules to ensure determinism of computation (as is proposed for example in [2]).

The second interesting extension of this work is strictness analysis. It has been introduced as a way of improving lazy systems by determining at compile time that some parts of the values are always needed and may be evaluated without "delaying" or "freezing" mechanism. If we know that the lazy pattern matching algorithm needs to evaluate some part of the argument of a function this gives us this kind of information. Moreover when compiling the expression associated with a pattern we can use the fact that some parts of the argument have been evaluated during the pattern matching process.

Strictness analysis has another connection with lazy pattern matching. In fact a lazy language does not need a lazy pattern matching algorithm : it only needs that the application of a function to its argument does not evaluate not needed part of the argument.

Now look at a function using Berry's example :

$$\begin{array}{lll} fun & (true, false, x) & \rightarrow \quad x \\ | & (false, x, true) & \rightarrow \quad x \\ | & (x, true, false) & \rightarrow \quad x \end{array}$$

Since we did not modify the patterns there is no lazy pattern matching algorithm, but as the expression to evaluate in order to return the result is in each case that part of the argument that the pattern matching algorithm does not need, we see that the whole argument is always needed. Hence the function's result may be lazyly evaluated using any pattern matching algorithm. However there is no mean of deciding what part of a general ML expression will always be evaluated. So there is here much work to gain efficiency from the concept of strictness analysis, and finding a lazy pattern matching algorithm remains the only systematic way to ensure that the function evaluation will always be lazy.

References

[1] L. Augustsson "A Compiler for Lazy ML", A.C.M. Conference on Lisp and Functional Programming, Austin 1984, pp 218-225

[2] J. Baeten J. Bergstra J. Klop "Priority Rewrite Systems", Report CS-R8407, Center for Mathematics and Computer Science, Amsterdam

[3] R. Burstall D. MacQueen D. Sannella "HOPE : An Experimental Applicative Language", A.C.M. Conference on Lisp and Functional Programming, Stanford 1980, pp 136-143

[4] L. Cardelli "Compiling a Functional Language", A.C.M. Conference on Lisp and Functional Programming, Austin 1984, pp 208-217

[5] P.L. Curien "Categorical Combinators, Sequential Algorithms and Functional Programming", Research Notes in Theoretical Computer Science, Pitman Publishing Ltd 1986

[6] G. Huet J.J. Lévy "Call by Need Computations in Non Ambiguous Linear Term Rewriting Systems", Rapport IRIA Laboria 359, August 1979

[7] G. Kahn G. Plotkin "Domaines concrets", Rapport IRIA Laboria 336, December 1978

[8] M. Mauny "Compilation des Langages Fonctionnels dans les Combinateurs Catégoriques, Application au langage ML", Thèse de 3ème cycle, Université Paris 7, 1985

[9] M. Mauny A. Suarez "Implementing Functional Languages in the Categorical Abstract Machine", A.C.M. Conference on Lisp and Functional Programming, Cambridge 1986, pp 266-278

[10] R. Milner "A Proposal for Standard ML", A.C.M. Conference on Lisp and Functional Programming, Austin 1984, pp 184-197

[11] G. Plotkin "Call-by-need, Call-by-value and the Lambda Calculus", T.C.S. Vol 1, pp 125-159, 1975

[12] A. Suarez "Une Implémentation de ML en ML", Thèse, Université Paris 7, to appear

[13] D. Turner "Miranda a Non Strict Functional Language with Polymorphic Types", in J.P. Jouannaud ed. Functional Programming Languages and Computer Architecture, L.N.C.S. 201, Springer Verlag 1985

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique